

A fast divide-and-conquer algorithm for indexing human genome sequences

Woong-Kee Loh^{a,*}, Yang-Sae Moon^b, Wookey Lee^c

^aDepartment of Multimedia, Sungkyul University

^bDepartment of Computer Science, Kangwon National University

^cDepartment of Industrial Engineering, Inha University

Abstract

Since the release of human genome sequences, one of the most important research issues is about indexing the genome sequences, and the suffix tree is most widely adopted for that purpose. The traditional suffix tree construction algorithms have severe performance degradation due to the memory bottleneck problem. The recent disk-based algorithms also have limited performance improvement due to random disk accesses. Moreover, they do not fully utilize the recent CPUs with multiple cores. In this paper, we propose a fast algorithm based on ‘divide-and-conquer’ strategy for indexing the human genome sequences. Our algorithm almost eliminates random disk accesses by accessing the disk in the unit of contiguous chunks. In addition, our algorithm fully utilizes the multi-core CPUs by dividing the genome sequences into multiple partitions and then assigning each partition to a different core for parallel processing. Experimental results show that our algorithm outperforms the previous fastest DIGEST algorithm by up to 3.5 times.

Keywords: human genome sequences, indexing, suffix tree, memory bottleneck problem, divide-and-conquer, parallel processing

1. Introduction

Due to recent advances in bio technology (BT), genome sequences of diverse organisms including human beings are collected into databases. The Human Genome Project (HGP), which had been initiated in 1990, released the human DNA sequences of approximately 3Gbp¹ size in 2003. Since the release, a lot of researches are under their way for harnessing the genome sequences. An essential research issue is about indexing large-scale genome sequences for efficient retrieving of genome subsequences of interest [1, 3, 7, 9, 11, 12, 14]. The suffix tree is most widely adopted for indexing genome sequences [3, 4, 5, 9, 11, 14]. In general, a suffix tree is created for a given string (or sequence) X and enables efficient exact matching and approximate matching on substrings of X [8]. We explain the suffix tree in more detail in Section 2.

A lot of algorithms have been proposed for efficient construction of the suffix tree. Ukkonen’s algorithm [15] is the most famous one which, given a string of length n , constructs the corresponding suffix tree in $O(n)$ time. The algorithm implicitly assumes that n is small enough so that the input string and the output suffix tree can be loaded in the main memory as a whole. However, genome sequences could be several million or billion times larger than the strings dealt with the traditional suffix tree construction algorithms such as Ukkonen’s algorithm. Moreover, the

suffix tree is about 10 ~ 60 times larger than the input sequence [3, 11, 14]. Hence, the application of Ukkonen’s algorithm for large-scale genome sequences should cause severe disk swap in and out, which is generally called *memory bottleneck problem* or *thrashing* [3, 4, 5, 9, 11, 14]. Actually, TOP-Q algorithm [4], an extension of Ukkonen’s algorithm, took seven hours for constructing the suffix tree for genome sequences of 40Mbp, which is much smaller than the human genome sequences, and it could not finish for genome sequences of 60Mbp [11].

For coping with the memory bottleneck problem, a few disk-based algorithms have been proposed for constructing the suffix tree [3, 5, 9, 11, 14]. Disks have much larger size than main memory at the lower cost; however, they require much longer access time up to several hundred times. Hence, the disk-based algorithms are designed mainly to maximize the main memory utilization and the disk access efficiency. However, these algorithms have a common drawback that they incur random disk accesses. The disk access performance is dependent more on access patterns than access amount; even for accessing the same amount, the random disk access requires much more time than the sequential disk access. Thus, the disk-based algorithms have been improved in the way of decreasing the ratio of random disk accesses.

Another problem of the previous disk-based algorithms is that they do not fully utilize the most up-to-date CPU technologies. Instead of raising the clock speed, recent CPUs are designed to have multiple, simultaneously running cores that enable intra-CPU parallel processing. How-

*Corresponding author

¹bp stands for ‘base pair.’ There are four bases, namely adenine (A), cytosine (C), guanine (G), and thymine (T).

ever, some previous algorithms run mostly on a single core, and the others suffer from severe interference among the threads and hence have little gain by parallel processing. We explain the problems of the previous algorithms in more detail in Section 3.

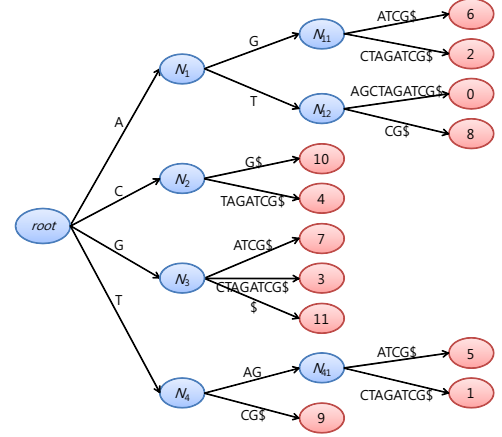
In this paper, we propose a fast algorithm based on ‘divide-and-conquer’ strategy for constructing the suffix tree for large-scale human genome sequences. The most significant difference from the previous algorithms is that the proposed algorithm almost eliminates random disk accesses by accessing the disk in the unit of contiguous chunks each of which stores an entire suffix subtree. In addition, our algorithm fully utilizes the multi-core CPUs by dividing the genome sequences into multiple, independent partitions and then assigning each partition to a different core for parallel construction of suffix subtrees. As an experimental result, our algorithm finished construction of the suffix tree for the entire human genome sequences in 64 minutes and outperformed DIGEST algorithm [3], which had previously been the fastest disk-based algorithm, by up to 3.5 times.

This paper is organized as the following. In Section 2, we briefly explain on the suffix tree. In Section 3, we explain on the previous disk-based suffix tree construction algorithms. We also explain the performance degradation by random disk accesses in the section. In Section 4, we propose a new disk-based suffix tree construction algorithm, and then in Section 5, we evaluate the performance of our algorithm through a series of experiments.

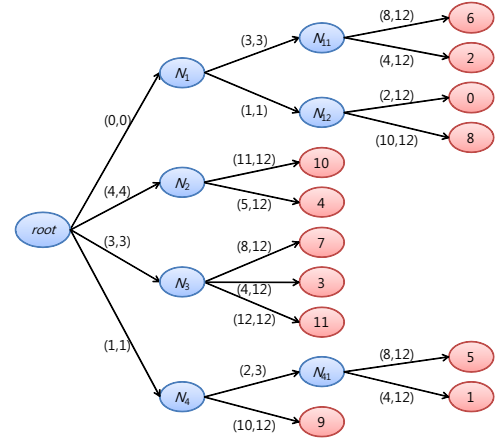
2. Suffix tree

Figure 1 shows the suffix tree for a short DNA sequence $X = \text{ATAGCTAGATCG\$}$. The symbol ‘\$’ is appended at the end of X so as to prohibit any suffix in X from being the prefix of any other suffix. Given a query sequence S , the search begins from the root node of the suffix tree. From the outbound edges of the root node, an edge e is chosen such that the label of e is the prefix of S . If no such edge is found, the search ends; if found, the child node N_e is visited by following the edge e , i.e., e is the inbound edge of N_e . Let l be the label length of e , $p_l(S)$ be the prefix of S of length l , and $s_l(S)$ be the suffix of S of length $\text{Len}(S) - l$. Then, it holds that $S = p_l(S) \oplus s_l(S)$, where \oplus is the sequence concatenation operator. The search for query subsequence $s_l(S)$ begins recursively at the node N_e in the same manner as the root node. The search goes on until a terminal node is reached in the suffix tree or there is no query (sub)sequence to be searched for.

Let us take a query sequence $S = \text{AGATCG}$ for example. In Figure 1(a), from the outbound edges of the root node, the edge with label ‘A’ is followed and then the node N_1 is visited. The search for query subsequence $s_l(S) = \text{GATCG}$ is performed recursively at the node N_1 . The search continues until the terminal node with position 6 is reached; it indicates that query sequence S is found at position 6 in the sequence X . Figure 1(b) shows the



(a) Edge labels are represented with subsequences.



(b) Edge labels are represented with (start, end) positions in X .

Figure 1: Suffix tree for a sequence $X = \text{ATAGCTAGATCG\$}$.

suffix tree whose edge labels are represented with (start, end) positions in X . While the labels’ representation sizes in Figure 1(a) are arbitrary, those in Figure 1(b) are all identical.

3. Related work

Hunt et al. [9] proposed the first disk-based suffix tree construction algorithm. Hunt’s algorithm excludes construction of suffix links, which caused severe memory bottleneck problem in Ukkonen’s algorithm [15]. Hunt’s algorithm divides the given genome sequences into partitions and then constructs a separate suffix subtree for each partition. Although Hunt’s algorithm has $O(n^2)$ complexity, it shows better indexing performance than Ukkonen’s algorithm by reducing disk accesses. However, Hunt’s algorithm incurs heavy random disk accesses since it stores each node in the suffix tree as a separate object using the persistent Java object storage interface called PJama [2]. Actually, the algorithm was successful in indexing genome sequences of up to 286Mbp size, but it could not be used for indexing the human genome sequences [9].

Tian et al. [14] presented the Top-Down Disk-based (TDD) approach for constructing disk-based suffix trees. TDD consists of two algorithms: Partition and Write Only Top Down (PWOTD) algorithm based on Wotd-eager algorithm [6] for constructing suffix trees and a memory buffer management algorithm for maximizing the performance of PWOTD algorithm. The performance of PWOTD algorithm highly depend on the settings of the memory buffer management algorithm [14]. Tian et al. [14] showed that TDD incurred only one sixth of disk accesses than DynaCluster algorithm [5], an extension of Hunt’s algorithm, and that TDD constructed the suffix tree for the entire human genome sequences in 30 hours. However, the memory buffer management algorithm in TDD assigns only a small portion of memory for keeping the suffix tree in main memory, while it assigns the largest portion to input genome sequences. TDD uses Least Recently Used (LRU) policy for swapping out the memory buffers into disk while constructing the suffix tree. Whenever PWOTD algorithm creates a new node N , it needs to access N ’s parent node P that could be previously stored far away from N . This causes random disk accesses, and the larger genome sequences should cause more random accesses.

Phoophakdee and Zaki [11] proposed an algorithm called TRELLIS, which eliminated data skewness among suffix subtrees by dividing genome sequences according to variable-length prefixes. Unlike Hunt’s algorithm [9] and TDD [14], TRELLIS can create suffix links optionally after the suffix tree is constructed. TRELLIS consists of three phases: prefix creation, partitioning, and merging phases. In the prefix creation phase, variable-length prefixes are created so that, for each prefix P_j , the suffix subtree T_j corresponding to the suffixes having the prefix P_j can be loaded into main memory as a whole. In the partitioning phase, the entire genome sequences are divided into partitions so that each partition R_i and its corresponding suffix tree T_i can be loaded into main memory as a whole. Then, a suffix tree T_i is constructed for each partition in this phase. In the merging phase, for each prefix P_j created in the prefix creation phase, the suffix subtrees $T_{i,j}$ are extracted from the suffix trees T_i and then merged into a single suffix subtree T_j . Phoophakdee and Zaki [11] showed that TRELLIS outperformed TDD by up to 4 times and that it constructed the suffix tree for the entire human genome sequences in 4.2 hours. However, since TRELLIS extracts the suffix subtrees $T_{i,j}$ stored at random positions in the suffix trees T_i in the merging phase, it incurs severe random disk accesses. Actually, the merging phase requires the longest execution time [11].

Ghoting and Makarychev [7] proposed an algorithm called WAVEFRONT based on ‘partition-and-merge’ strategy as TRELLIS [11]. WAVEFRONT divides the entire data into I/O-efficient partitions and processes each partition independently. In [7], WAVEFRONT was extended to be executed on a massively parallel system. The algorithm completed indexing the entire human genome sequences in 15 minutes on IBM Blue Gene/L system composed of

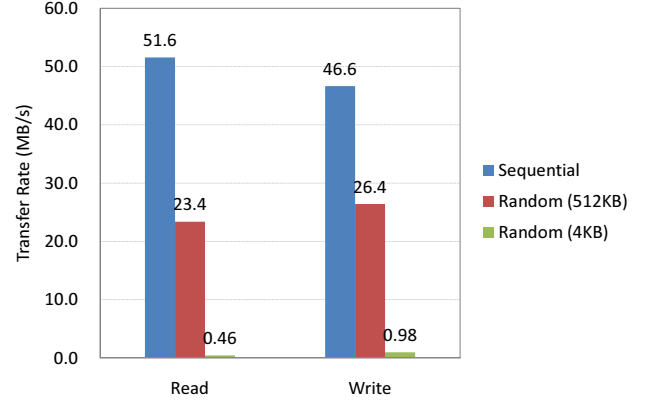


Figure 2: Disk read/write transfer rates: sequential read/write performed much better than random read/write.

1024 processors [7]. However, WAVEFRONT executed on a single processor showed no noticeable performance improvement compared with TRELLIS [11].

Barsky et al. [3] proposed an algorithm called DIGEST which consists of two phases similar to the merge-sort algorithm. In the first phase, the entire genome sequence is divided into partitions of the same length so that each partition can be loaded into main memory. For each partition, the suffixes contained therein are sorted in main memory and then are stored in disk. In the second phase, the suffixes sorted separately in each partition are merge-sorted. Suffix blocks from each partition are read sequentially one by one into main memory. The suffixes in different blocks are compared with each other, and the smallest one is extracted and then saved in the output block. When the output block becomes full, it is stored in disk. This continues until all the input blocks are empty. The sorted suffixes is called a *suffix array*, and it is known that a suffix array can be easily converted into a suffix tree [3, 13]. Barsky et al. [3] showed that DIGEST outperformed TRELLIS+ [12], an extension of TRELLIS [11], by up to 40% and that the algorithm completed indexing the entire human genome sequences in about 85 minutes. However, DIGEST should read suffix blocks from each partition stored at random positions in the second phase and hence suffers from severe random disk accesses. Moreover, since the merging phases of TRELLIS and DIGEST cannot be parallelized, they have little performance gain even by using recent multi-core CPUs.

As explained so far, the common drawback of the previous algorithms is the performance degradation due to random disk accesses. Figure 2 shows an experimental result of reading/writing a disk volume of 100MB size. The volume was read and written sequentially and at random in the unit of 512KB and 4KB. In the figure, the sequential read/write performed up to 112.1 and 47.7 times better than random read/write, respectively. The values in Figure 2 should be different according to experimental environments, though it is always the case that sequential accesses have better performance than random accesses.

4. Proposed indexing algorithm

In this section, we propose a new algorithm for indexing human genome sequences. The human genome is composed of 46 chromosomes: 22 chromosome pairs numbered 1 ~ 22 and x/y (sex) chromosomes. In this paper, we concatenate the entire genome sequences into a single long sequence and use this sequence as the input of our algorithm. This helps simplify indexing and searching algorithms.

Our algorithm is designed based on divide-and-conquer strategy: it divides the entire human genome sequence into multiple independent partitions and then constructs the suffix tree separately for each partition. The suffix tree for each partition is constructed in a contiguous chunk in main memory. When the construction is completed, the chunk image is stored sequentially into disk as it is. Hence, unlike TRELLIS and DIGEST [3, 11], our algorithm has no performance degradation due to random disk accesses. Moreover, since the suffix trees for different partitions are constructed independently and are not merged thereafter, their construction can be done in parallel by fully utilizing the most up-to-date multi-core CPUs. According to these features, our algorithm achieves dramatic performance improvement compared with the previous algorithms.

Our algorithm represents each base as a 2-bit code as in [3, 11, 12, 16]; A, C, G, and T are represented as 00, 01, 10, and 11, respectively. Since the human genome sequence has the size of approximately 3Gbp, the 2-bit coded sequence has the size of about $3\text{Gbp} / 4 = 750\text{MB}$. Actually, after removing unidentified base pairs, the 2-bit coded sequence has the size of about 700MB and can be fully loaded in main memory. Our algorithm assigns memory region for the full 2-bit coded genome sequence at the beginning and retains it to the end.

Our algorithm divides the human genome sequence into partitions according to prefixes, i.e., the suffixes having the common prefix belong to the same partition. We explain how to determine the prefixes for partitioning at the end of this section. The partitions are not necessarily created by physically dividing the genome sequence, but only the suffix positions are managed for each partition. While scanning the entire genome sequence, our algorithm creates the lists of suffix positions simultaneously for every prefix determined earlier; the list for a prefix P_j ($0 \leq j < m$) contains the positions of suffixes having the prefix P_j , where m is the number of partitions. Although each of these lists has a small size, the entire lists occupy a considerable amount of memory. Hence, the lists are stored in disk right after their creation; each list is retrieved from disk only once when the suffix tree is about to be constructed for the corresponding partition. Our algorithm creates each list of suffix positions in a contiguous memory region to read/write the list with a single operation and hence to eliminate random disk accesses. To obtain the sizes of contiguous memory regions, our algorithm scans the human genome sequence to count the frequency of every prefix before creating the lists of suffix positions.

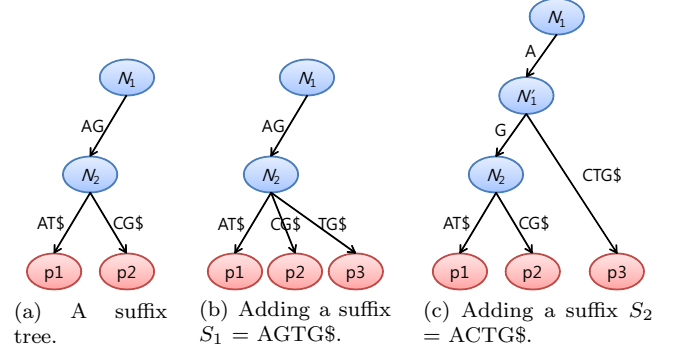


Figure 3: Example of adding suffixes into a suffix tree.

When the creation of partitions (i.e., the lists of suffix positions in the human genome sequence) is completed, our algorithm constructs the suffix tree separately for each partition. At first, our algorithm creates an empty suffix tree without any node and then adds suffixes one by one into the suffix tree while scanning the corresponding list of suffix positions. Figure 3 shows an example of adding suffixes into a suffix tree. Figure 3(a) shows a suffix tree before addition. Figure 3(b) shows the result of adding a suffix $S_1 = \text{AGTG\$}$ into the suffix tree in Figure 3(a). S_1 has the prefix $p_2(S_1) = \text{AG}$ of length 2 which matches the label of the outbound edge of N_1 and then $s_2(S_1) = \text{TG\$}$ does not have common prefix with any label of the outbound edges of N_2 . In this case, our algorithm creates a new outbound edge e of N_2 and labels it with $s_2(S_1) = \text{TG\$}$. The edge e is connected to a new terminal node p_3 , i.e., e becomes the inbound edge of p_3 . Figure 3(c) shows the result of adding a suffix $S_2 = \text{ACTG\$}$ into the suffix tree in Figure 3(a). The label of the outbound edge of N_1 partially matches the prefix $p_1(S_2) = \text{A}$ of S_2 . In this case, our algorithm cuts the outbound edge of N_1 and adds a new internal node N_1' ; the inbound edge of N_1' has the label $p_1(S_2) = \text{A}$. A new outbound edge e is added to node N_1' and is labeled with $s_1(S_2) = \text{CTG\$}$. The edge e is connected to a new terminal node p_3 , i.e., e becomes the inbound edge of p_3 .

Each time a suffix is added into the suffix tree, a new terminal node is created in the tree. Since every suffix ends with the symbol $\$$, the suffix cannot be a prefix of any other suffixes and has a unique position in the human genome sequence. Hence, a terminal node should exist in the suffix tree for representing the unique position of each suffix. The terminal node should have an inbound edge in the tree. The edge is an outbound edge of either (1) an existing node (Figure 3(b) case) or (2) a new node added between the cut edges (Figure 3(c) case). There exist no other cases.

Figure 4 shows the generalization of adding suffixes into the suffix tree by our algorithm. Let us assume that we have visited the node N_i in the course of searching for a suffix S in Figure 4(a). The concatenation $L = L_1 \oplus \dots \oplus L_i$ of edge labels from the root node to N_i should be the same

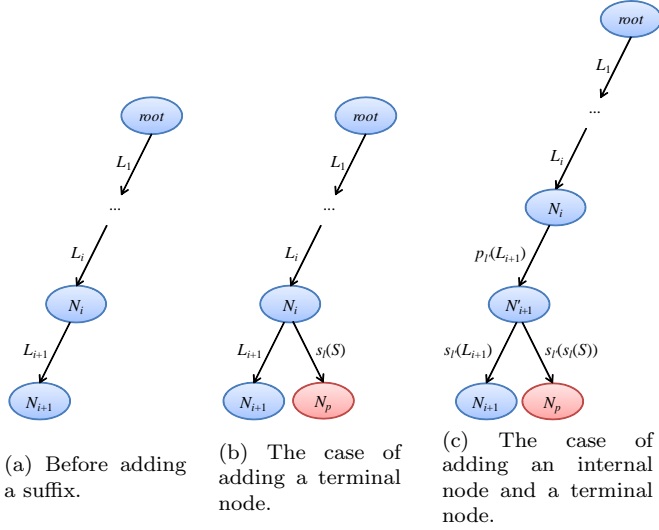


Figure 4: Generalization of adding suffixes into a suffix tree.

a	b	right	foo	misc
(unsigned int)	(unsigned int)	(unsigned int)	(unsigned int)	(unsigned short)

Figure 5: Data structure of our algorithm: the information on a node and its inbound edge is contained together.

as the prefix $p_l(S)$ of length $l = \text{Len}(L)$, i.e., $L = p_l(S)$. In case $L_{i+1} \cap s_l(S) = \emptyset$, an edge e labeled with $s_l(S)$ and a new terminal node N_p with the inbound edge e are added as in Figure 4(b). In case $L_{i+1} \cap s_l(S) = L' (\neq \emptyset)$, a new internal node N'_{i+1} and a new terminal node N_p are added as in Figure 4(c), where $l' = \text{Len}(L')$ and $p_{l'}(L_{i+1}) = p_{l'}(s_l(S)) = L'$. Since the suffix always ends with \$, we cannot have the case $S = L$ in Figure 4.

Figure 5 shows the data structure of our algorithm. As shown in the figure, the information on a node and its inbound edge is contained together in a single data structure. The fields a and b represent the start and end positions of the inbound edge in the human genome sequence as shown in Figure 1(b). The field $right$ contains the pointer to the next sibling node, and foo represents either (1) a pointer to the leftmost child node in case of an internal node or (2) the suffix position in the genome sequence in case of a terminal node. The field $misc$ contains miscellaneous information on the node. The fields a , b , $right$, and foo are 4-byte unsigned integers, while the field $misc$ is a 2-byte unsigned integer. Hence, the data structure has the fixed length of 18 bytes. For distinguishing between the internal and terminal nodes, the field b is investigated. If $b = n$, where n is the length of genome sequence, it is a terminal node; if $b < n$, it is an internal node (refer to Figure 1(b)).

We can efficiently construct the suffix trees using the data structure in Figure 5. We explain this using Figure 6, which shows the representation of suffix trees in Figure 4 using the data structure; Figures 6(a) ~ 6(c) correspond to Figures 4(a) ~ 4(c), respectively. In Figure 6(a), the

fields (a_i, b_i) and (a_{i+1}, b_{i+1}) represent the start and end positions of labels L_i and L_{i+1} , respectively. The fields with X stand for “don’t care” fields, which are not used nor updated here. The arrow indicates a pointer to a possible distant node. The nodes N_i and N_{i+1} may not be adjacent as shown in the figure, though N_{i+1} is easily accessed by following the pointer. Figure 6(b) shows the case a new terminal node N_p is added. The node N_{i+1} can be either an internal or a terminal node and is a sibling node of N_p . In the figure, the leftmost child node of N_i has been changed from N_{i+1} to N_p . This is because we can efficiently add N_p as a new child node of N_i without accessing N_{i+1} and all its sibling nodes. Figure 6(c) shows the case a new internal node N'_{i+1} and a new terminal node N_p are added. The field values of the N_{i+1} are copied to the newly allocated node region, and then the field a_{i+1} is adjusted (b_{i+1} is not changed). The field values of N'_{i+1} are set in the region previously used by N_{i+1} as shown in the figure. The node N_p is a sibling node of N_{i+1} and is added as the leftmost child node of N'_{i+1} as in Figure 6(b). The key idea we would like to show in Figure 6 is that, when a suffix is added, there is only slight modification in the suffix tree constructed so far; it can be done only by allocating new memory region(s) for one or two nodes and then setting a few appropriate field values therein. This is one of the features providing the efficiency of our algorithm.

Our algorithm constructs a suffix tree in a main memory chunk. Allocations of memory regions for new nodes (and their inbound edges) are made sequentially in the chunk. The pointers in Figures 5 and 6 are relative offset values from the beginning of the chunk. Once the construction of a suffix tree is completed, our algorithm stores the chunk image into disk without any modification. When the chunk image is reloaded into main memory, the pointers are still valid regardless of where it is reloaded. Since the chunk image is stored in and read from the disk sequentially, there is no performance degradation due to random disk accesses, and thus we have significantly improved performance. When multiple suffix trees are constructed in parallel, our algorithm allocates a separate memory chunk for each suffix tree. Even in this case, the human genome sequence is loaded only once into the memory region shared by the simultaneous processes of our algorithm. This parallel processing enables more significant performance improvement.

We now explain how to determine the prefixes for dividing the human genome sequence into partitions. Each suffix in the genome sequence is assigned to a partition according to its prefix; every suffix in a partition has a common prefix. Given a prefix length p , our algorithm creates a partition for each possible prefix of length p . The number of partitions is 4^p . A weakness of this scheme is that it causes data skewness among the partitions [11]; there may be big differences among the sizes of partitions and hence the corresponding suffix trees. We tackle this weakness as follows. As p increases, the number of suffixes in each par-

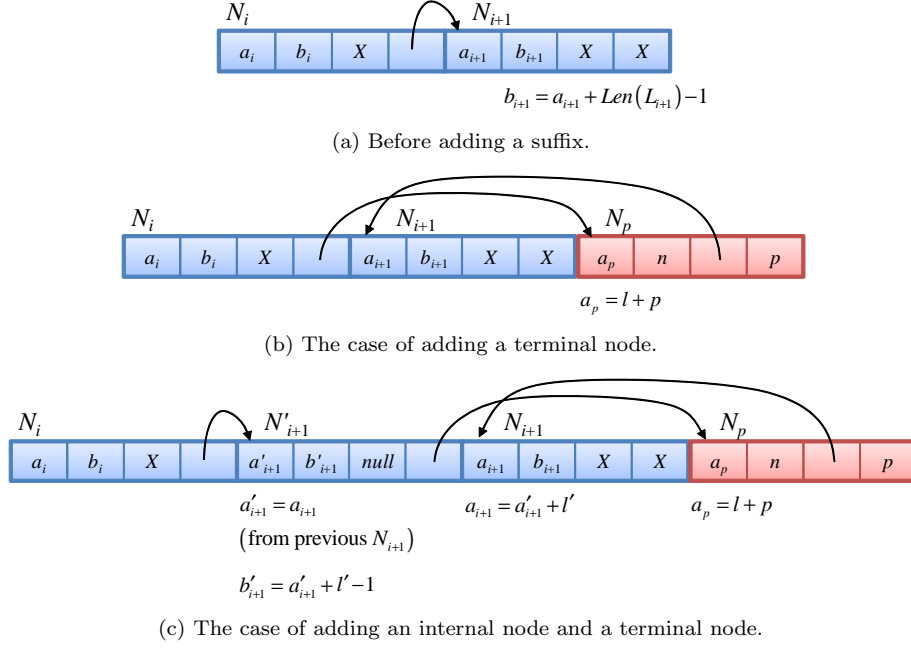


Figure 6: Data structures corresponding to the suffix trees in Figure 4.

tition decreases, and the size of corresponding suffix tree also decreases. We set p to be large enough to make the suffix tree sizes smaller than the size M of available main memory. Then, the simultaneous processes of our algorithm choose the partitions so that the estimated sizes of their corresponding suffix trees sum up very close to M . This can be done with simple computations. By fully utilizing main memory in this way, our algorithm achieves better indexing performance.

The minimum length of prefixes is computed approximately using the following Eq. (1):

$$p_{\min} = \left\lceil \log_4 \frac{n \cdot f}{M} \right\rceil, \quad (1)$$

where n is the length of human genome sequence and f is a multiplication factor to estimate the suffix tree size. M represents the size of remaining main memory after loading the entire 2-bit coded human genome sequence. f is defined as the maximum of $\frac{T}{s}$, where s is the length of a genome sequence and T is the size of the corresponding suffix tree. We estimate the size of a big suffix tree by test construction of small suffix trees. The f value greatly differs according to suffix tree construction algorithms and is about $30 \sim 32$ in our algorithm.

5. Performance evaluation

In this section, we show the superiority of our algorithm through a series of experiments. We use the same data sets as those in [3]. The first set is a short genome sequence of 110Mbp size obtained from 6643 organisms. The second set is the entire human genome sequence of about 3Gbp

size. These data sets are denoted as VDB and HG18, respectively.

The hardware platform is a PC equipped with Intel Core2Quad Q9550 2.83GHz CPU, Samsung DDR3 8GB main memory, and a 500GB 7200rpm hard disk. The software platforms are Ubuntu 10.10 32bit Linux and Windows 7 64bit Edition. The first experiment was performed on Ubuntu as in [3], and the second and third experiments were performed on Windows 7. The latter two experiments were also performed on Ubuntu, though we had 10 ~ 15% better performance on Windows 7. As C/C++ compilers, we used GNU C++ 4.4.5 on Ubuntu and Visual C++ 2010 Express Edition on Windows 7.

In the first experiment, we compared the performance of our algorithm with DIGEST [3], which had been the fastest disk-based suffix tree construction algorithm. We downloaded the source code of DIGEST from the author's web site². In this experiment, we ran our algorithm and DIGEST on VDB data set and compared their elapsed time for constructing the suffix trees³. Figure 7 shows the result of experiment; our algorithm outperformed DIGEST by up to 3.5 times. We executed only one process of our algorithm in this experiment. If we had executed multiple parallel processes of our algorithm, we could have achieved higher performance improvement.

In the second experiment, we ran our algorithm on both VDB and HG18 data sets and compared the elapsed

²<http://webhome.cs.uvic.ca/~mgbarsky/>

³We also tried the experiment on HG18 data set; however, DIGEST always terminated abnormally with the segmentation fault error. We discussed on this with the author of DIGEST, but we could not solve the problem to the end.

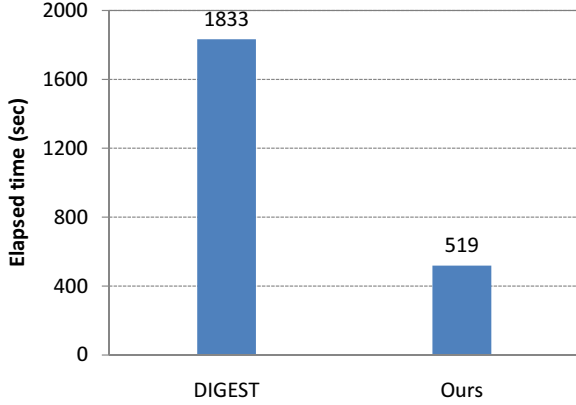


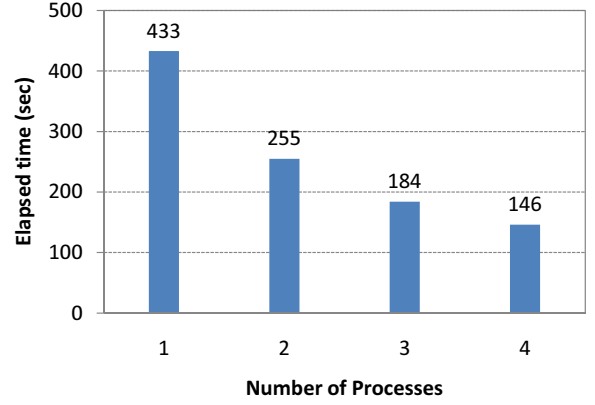
Figure 7: Result of first experiment: our algorithm outperformed DIGEST by up to 3.5 times.

time for various numbers of parallel processes of our algorithm. Figure 8 shows the experimental result. Since the hardware platform has a four-core CPU, we increased the number of parallel processes up to four. Actually, we could have almost no performance improvement by running more than four parallel processes on the same platform. Note that the units of vertical axes are seconds and minutes in Figures 8(a) and 8(b), respectively. As shown in the figures, we obtained performance improvement by up to 3.0 times by running four parallel processes compared with a single process. We could not obtain four times performance improvement mostly due to inter-process communication and synchronization. Since our algorithm is designed to minimize the effect of disk accesses, it has high potential of more performance improvement by using the advanced CPUs with more cores and faster clock speeds.

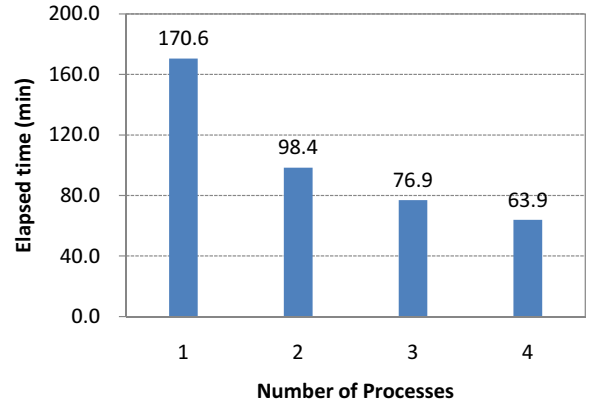
In the third experiment, we measured the elapsed time of our algorithm for various sizes of genome sequences. We ran four processes on the genome sequences consisting of the first 2, 5, 8, 11, 15, and 24 chromosomes in the human genome sequence. Figure 9 shows the result. As the result of regression analysis on the experimental result, we could find that the elapsed time is almost linearly correlated with the size of genome sequences.

References

- [1] S. Altschul, T. Madden, A. Schaffer, J. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," *Nucleic Acids Research*, Vol. 25, No. 17, pp. 3389-3402, 1997.
- [2] M. Atkinson and M. Jordan, "Providing Orthogonal Persistence for Java," In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, pp. 383-395, July 1998.
- [3] M. Barsky, U. Stege, A. Thomo, and C. Upton, "A new method for indexing genomes using on-disk suffix trees," In *Proc. ACM Conference on Information and knowledge Management (CIKM)*, Napa Valley, California, pp. 649-658, Oct. 2008.
- [4] S. J. Bedathur and J. R. Haritsa, "Engineering a Fast Online Persistent Suffix Tree Construction," In *Proc. Int'l Conf. on Data Engineering (ICDE)*, IEEE, Boston, Massachusetts, pp. 720-731, Mar. 2004.



(a) Using VDB data set.



(b) Using HG18 data set.

Figure 8: Result of second experiment: we could obtain performance improvement by up to 3.0 times by running four parallel processes.

- [5] C.-F. Cheung, J. Yu, and H. Lu, "Constructing Suffix Tree for Gigabyte Sequences with Megabyte Memory," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, Vol. 17, No. 1, pp. 90-105, Jan. 2005.
- [6] R. Giegerich, S. Kurtz, and J. Stoye, "Efficient Implementation of Lazy Suffix Trees," *Software: Practice and Experience (SPE)*, Vol. 33, No. 11, pp. 1035-1049, 2003.
- [7] A. Ghoting and K. Makarychev, "Serial and parallel methods for i/o efficient suffix tree construction," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Providence, Rhode Island, pp. 827-840, June 2009.
- [8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
- [9] E. Hunt, M. P. Atkinson, and R. W. Irving, "Database Indexing for Large DNA and Protein Sequence Collections," *The VLDB Journal*, Vol. 11, No. 3, pp. 256-271, 2002.
- [10] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," *Software: Practice and Experience (SPE)*, Vol. 29, No. 13, pp. 1149-1171, Nov. 1999.
- [11] B. Phoophakdee and M. J. Zaki, "Genome-scale Disk-based Suffix Tree Indexing," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, pp. 833-844, Beijing, China, June 2007.
- [12] B. Phoophakdee and M. J. Zaki, "TRELLIS+: An Effective Approach for Indexing Genome-Scale Sequences Using Suffix Trees," In *Proc. Pacific Symp. on Biocomputing*, pp. 90-101, Kohala Coast, Hawaii, Jan. 2008.
- [13] R. Sinha, S. Puglisi, A. Moffat, and A. Turpin, "Improving suffix array locality for fast pattern matching on disk," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Vancouver, Canada, pp. 661-672, June 2008.
- [14] Y. Tian, S. Tata, R. A. Hankins, and J. M. Patel, "Practical

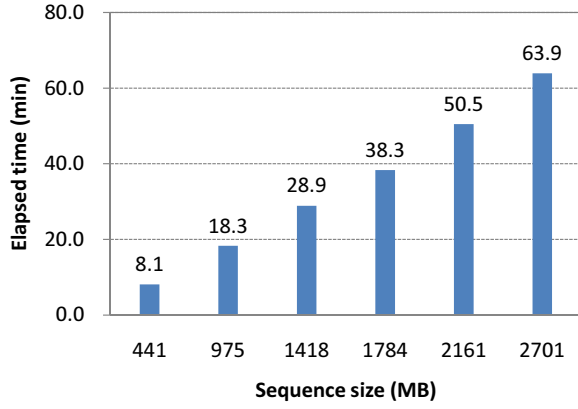


Figure 9: Result of third experiment: elapsed time has the linear correlation with the size of genome sequences.

- Methods for Constructing Suffix Trees,” *The VLDB Journal*, Vol. 14, No. 3, pp. 281-299, 2005.
- [15] E. Ukkonen, “On-line Construction of Suffix Trees,” *Algorithmica*, Vol. 14, No. 3, pp. 249-260, Sept. 1995.
- [16] J.-I. Won, S.-K. Hong, J.-H. Yoon, S. Park, and S.-W. Kim, “A Practical Method for Approximate Subsequence Search in DNA Databases,” In *Proc. Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD)*, Nanjing, China, pp. 921-931, May 2007.